Aegir

Date: October 24, 1997

PROGRESS REPORT

on the Project
Automatic Target Recognition (N94-124)

for
SMALL BUSINESS INNOVATION RESEARCH
Phase II

Aegir Contract No.: 1079-000
Contract No.: N00014-96-C-0069
CDRL No. 0001

Prepared by:

Aegir Systems, Inc.
2051 N. Solar Drive; Suite 200
Oxnard, CA 93030
(805)485-4888

for
Program Officer
Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217-5660
Attn: Julia Abrahams Code 311
Ref: Contract N0014-96-C-0069

Prepared by:

B. Rozovsky, C. Rao (Filimage)
Author

Approved by:

Program Manager

Data Manager

Fast Optimal Nonlinear Filter: A Software for 3-D Stochastic Dynamic Systems

Abstract

In this report, a real-time optimal nonlinear filter is developed. This fast filter is based on two techniques: (i) splitting of the convection and diffusion operators, and (ii) tracking of important windows. Presented is an explicit scheme using the forward characteristic equation and the fast Gauss transform. The domain of interest is determined adaptively. Subroutines of the software code are briefly described and numerical results for 3-D problems are given.

Contents

1 Introduction

Nonlinear filtering is the process of computing estimates of the current state $\boldsymbol{x}_t$ of a nonlinear stochastic, or noisy, dynamic system (e.g., a rapidly maneuvering target), given the current measurements $\boldsymbol{z}_k$ which are obtained as nonlinear functions of the system states plus noise. The states may include such unknown target characteristics as position, speed, acceleration, aspect angle, etc. The relationship between measurements and target states is modeled by a (generally nonlinear) measurement equation of the form $\boldsymbol{z}_k = h(\boldsymbol{x}_k, \boldsymbol{v}_k)$ where $\boldsymbol{v}_k$ denotes the noise process. The expected range of possible behaviors of the target is modeled by Markov-state transition equations of the form $\boldsymbol{x}_t = b(\boldsymbol{x}_t, \boldsymbol{w}_t)$ where $\boldsymbol{w}_t$ is another noise process.

There are military situations in which it is desirable to estimate the position, velocity and perhaps acceleration of a target from measurements of angle but not range. A well-known example is the determination by a submarine of planar position and velocity of a

ship from passive sonar measurements, because the submarine commander does not want to reveal his presence by pinging. In air warfare, a fighter defending against a raid may wish to launch a missile against a jammer at unknown range, but should not do so unless the jammer's position and velocity can be estimated. A more recent problem is estimation of target position, velocity and acceleration in three dimensions from angle measurements only, either with a passive IR receiver or a jammed radar receiver on a missile, in order to utilize optimal guidance.

The extended Kalman filter (EKF) has been the main choice in most approaches to angle-only tracking. But a serious limitation of EKF application to angle-only tracking is the nontrivial nonlinearity of the underlying problem. Its mathematical model in 3D can be described as follows.

Signal:

$$\begin{aligned} dx_1(t) &= b_1(x_1(t), x_2(t), x_3(t))dt + \sigma_1 dw_1(t), \\ dx_2(t) &= b_2(x_1(t), x_2(t), x_3(t))dt + \sigma_2 dw_2(t), \\ dx_3(t) &= b_3(x_1(t), x_2(t), x_3(t))dt + \sigma_3 dw_3(t); \end{aligned}$$

Observation:

$$\begin{aligned} z_1(k) &= \text{sign}(x_2(t_k)) \arccos \tfrac{x_1(t_k)}{x_1^2(t_k)+x_2^2(t_k)} + \varepsilon_1(k)v_1(k), \\ z_2(k) &= \arcsin \tfrac{x_3(t_k)}{x_1^2(t_k)+x_2^2(t_k)+x_3^2(t_k)} + \varepsilon_2(k)v_2(k), \end{aligned}$$

where $w_1(t), w_2(t), w_3(t)$ are independent Wiener processes, $v_1(k), v_2(k)$ are standard normal random variables, and the distribution of the initial state $x_1(0), x_2(0), x_3(0)$ is given. (Here we consider dynamic systems with discrete observations because in most cases the observational measurements are only available at discrete time moments.)

The modified polar coordinates (MPC) introduced by Hoelzer et al [9] are designed to reduce the nonlinear observations to linear ones. Unfortunately in doing so MPC also transforms the signal process. Unless the latter is of very simple nature, the MPC transform makes the signal system practically intractable.

Much more perspective approach to angle-only tracking is based on optimal nonlinear filtering. The optimal nonlinear filtering theory allows to compute the posterior density function $\pi_k(x)$ of the signal process $x(t_k) = (x_1(t_k), x_2(t_k), x_3(t_k))$ given observations $z(j)$, $j \leq k$. The posterior density function is defined by $\pi_k(x) = p_k(x)/\int p_k(\xi)d\xi$ where the so called unnormalized filtering density $p_k(x)$ can be formulated in terms of solutions of the corresponding Fokker-Planck equation (see the next section). For a long period of time practical application of nonlinear filtering has been strained by numerical difficulties related to on-line solution of the Fokker-Planck equation.

In this report, we propose a new algorithm for the target tracking problem, mainly for solving the corresponding Fokker-Planck equation. It has been demonstrated that the techniques used in the new algorithm decrease prediction error and provide a more accurate representation of the target bearings as compared to EKF. On the other hand, the algorithm is also fast, in that its calculations need only $O(N)$ flops per time step where N is the number of points in the spatial domain. An adaptive method is used to update the domain of interest so that the number of points in the domain is kept relatively small. Thus, our approximation of the optimal nonlinear filter has an optimal computational complexity for arbitrary nonlinear systems.

In Section 2, we state the nonlinear filtering problem in a more general and rigorous setting and formulate its theoretical solution. Sections 3 and 4 are devoted to the development of the fast nonlinear filter, and Section 5 to a brief documentation of the subroutines of the software. Finally numerical examples are given in Section 6.

2 Statement of the Nonlinear Filtering Problem

Now consider the dynamical system described by the stochastic differential equation

$$\begin{aligned} &dX(t) = b(X(t))dt + \sigma(X(t))dW(t), \quad t > 0, \\ &X(0) \sim \pi_0(x), \end{aligned} \tag{1}$$

and the discrete observations given by

$$z(k) = h(X(t_k)) + \varepsilon(X(t_k))V(k), \quad k = 1, 2, \cdots, \tag{2}$$

where $\pi_0 : I\!R^n \to I\!R$ is the initial density, $b : I\!R^n \to I\!R^n$, and $h : I\!R^n \to I\!R^m$ are known vector-valued functions, $\sigma : I\!R^n \to I\!R^{n\times n}$ and $\varepsilon : I\!R^n \to I\!R^{m\times m}$ are known matrix-valued functions, $\{W(t)\}_{t\geq 0}$ is a standard n-dimensional Brownian motion, $\{V(k)\}_{k\geq 1}$ is a standard m-dimensional white Gaussian sequence, and $t_k = k\Delta$ ($\Delta > 0$). $X(0)$, $\{W(t)\}$ and $\{V(k)\}$ are assumed to be independent, and functions $b, h, \sigma, \varepsilon$ and π_0 are assumed to be smooth enough (satisfying certain regularity conditions, see [14][17]).

Let $f = f(x), x \in I\!R^n$, be a measurable scalar function such that $I\!E|f(X(t))|^2 < \infty$ for all $t \geq 0$. Then the filtering problem for (1)-(2) can be stated as follows: find the minimum variance estimate of $f(X(t_k))$ given the measurements $z(1), \cdots, z(k)$. This estimate is called *the optimal filter* and is known to be

$$\hat{f}(k) = I\!E\Big[f(X(t_k)) \mid z(1), \cdots, z(k)\Big].$$

The optimal filter can be characterized as follows.

Denote by T_t the solution operator for the Fokker-Planck equation corresponding to the state process; in other words, $u(t,x) = T_t\varphi(x)$ is the solution of the equation

$$\begin{aligned} \frac{\partial u(t,x)}{\partial t} &= \frac{1}{2}\sum_{\mu,\nu=1}^{n} \frac{\partial^2}{\partial x_\mu \partial x_\nu}\Big((\sigma(x)\sigma(x)^T)_{\mu\nu}u(t,x)\Big) \\ &\quad - \sum_{\nu=1}^{n}\frac{\partial}{\partial x_\nu}\Big(b_\nu(x)u(t,x)\Big), \quad t > 0, \\ u(0,x) &= \varphi(x), \end{aligned} \tag{3}$$

where $(\sigma(x)\sigma(x)^T)_{\mu\nu} = \sum_{i=1}^{m}\sigma_{\mu i}(x)\sigma_{\nu i}(x)$ is the μ-th row and ν-th column entry of $\sigma(x)\sigma(x)^T$, and $b_\nu(x)$ is the ν-th component of $b(x)$.

Define the *unnormalized filtering density* $p_k(x)$, for $x \in I\!R^n$ and $k \geq 0$, by

$$\begin{aligned} p_0(x) &= \pi_0(x), \\ p_k(x) &= \alpha_k(x)T_\Delta p_{k-1}(x), \end{aligned} \tag{4}$$

where

$$\alpha_k(x) = \exp\Big\{-\frac{1}{2}(z(k)-h(x))^T(\varepsilon(x)\varepsilon(x)^T)^{-1}(z(k)-h(x))\Big\}.$$

Then the optimal filter $\hat{f}(k)$ can be written as ([10])

$$\hat{f}(k) = \frac{\int_{\mathbb{R}^n} p_k(x)f(x)dx}{\int_{\mathbb{R}^n} p_k(\xi)d\xi}. \tag{5}$$

Remark If we have an initial observation z_0 at time $t = 0$, we can modify the initial (unnormalized) filtering density in (4) in the following way:

$$p_0(x) = \alpha_0(x)\pi_0(x).$$

3 Splitting of Convection and Diffusion Terms

Our objective here is to develop recursive numerical algorithms for computing the optimal filter in which the on-line computation is as simple as possible; in particular, the number of computer operations at each time step should be proportional to the number of grid points where the filtering density is numerically defined. The starting point in the derivation is the equation for the unnormalized filtering density in the general nonlinear model, and the approach is based on the technique known as operator splitting.

To compute the unnormalized filtering density, a fast Fokker-Planck solver is needed. In this section we present a method which is based on the operator-splitting technique ([11][15]). Similar ideas have been used in [8] for solving the Zakai equation arising from image filtering.

We first assume that in the noise term of (1) and the covariance matrix σ is constant and diagonal: $\sigma_{\mu\nu}(x) = \delta_{\mu\nu}a_\nu$. Then the Fokker-Planck equation (3) becomes

$$\begin{aligned}&\frac{\partial u(t,x)}{\partial t} = \frac{1}{2}\sum_{\nu=1}^{n}\frac{\partial^2}{\partial x_\nu^2}\Big(a_\nu^2 u(t,x)\Big) - \sum_{\nu=1}^{n}\frac{\partial}{\partial x_\nu}\Big(b_\nu(x)u(t,x)\Big), \quad t>0,\\ &u(0,x) = \varphi(x)\,.\end{aligned}$$

Its solution can be expressed as

$$T_t\varphi(x) = I\!E\Big[\varphi(\xi^x(t))\exp\Big\{-\int_0^t(\nabla\cdot b)(\xi^x(s))ds\Big\}\Big]\,, \tag{6}$$

where $\xi^x(t)$ is a stochastic process satisfying

$$\begin{aligned}&d\xi^x(t) = -b(\xi^x(t))dt + \mathrm{diag}(a_\nu)dW(t)\,,\\ &I\!P[\xi^x(0) = x] = 1\,.\end{aligned}$$

To proceed the splitting of convection and diffusion terms, denote by T_t^c and T_t^d the solution operators of the equations

$$\begin{aligned}&\frac{\partial u(t,x)}{\partial t} = -\sum_{\nu=1}^{n}\frac{\partial}{\partial x_\nu}\Big(b_\nu(x)u(t,x)\Big), \quad t>0,\\ &u(0,x) = \varphi(x)\,,\end{aligned}$$

and

$$\frac{\partial u(t,x)}{\partial t} = \frac{1}{2}\sum_{\nu=1}^{n}\frac{\partial^2}{\partial x_\nu^2}\big(a_\nu^2 u(t,x)\big), \quad t>0,$$
$$u(0,x) = \varphi(x)\,,$$

respectively. In fact, the two solution operators can be expressed explicitly as

$$T_t^c\varphi(x) = \varphi(\eta^x(t))\exp\Big\{-\int_0^t(\nabla\cdot b)(\eta^x(s))ds\Big\}\,, \tag{7}$$

and

$$T_t^d\varphi(x) = I\!E[\varphi(\zeta^x(t))] = \frac{(\pi t)^{-n/2}}{a_1\cdots a_n}\int_{I\!R^n}\exp\Big\{-\sum_{\nu=1}^{n}\frac{(x_\nu-y_\nu)^2}{2a_\nu^2 t}\Big\}\varphi(y)dy\,, \tag{8}$$

where processes $\eta^x(t)$ (deterministic) and $\zeta^x(t)$ satisfy

$$d\eta^x(t) = -b(\eta^x(t))dt, \quad \eta^x(0) = x\,,$$

and

$$d\zeta^x(t) = \mathrm{diag}(a_\nu)dW(t), \quad \zeta^x(0) = x\,.$$

Then it can be shown that the following approximation formulas hold:

$$T_\Delta\varphi = T_\Delta^d T_\Delta^c\varphi + O(\Delta^2)\,, \tag{9}$$
$$T_\Delta\varphi = T_\Delta^c T_\Delta^d\varphi + O(\Delta^2)\,, \tag{10}$$
$$T_\Delta\varphi = T_{\frac{\Delta}{2}}^c T_\Delta^d T_{\frac{\Delta}{2}}^c\varphi + O(\Delta^3)\,, \tag{11}$$
$$T_\Delta\varphi = T_{\frac{\Delta}{2}}^d T_\Delta^c T_{\frac{\Delta}{2}}^d\varphi + O(\Delta^3)\,. \tag{12}$$

Therefore, instead of solving the original Fokker-Planck equation, we only need to compute (7) and (8). To compute (7), we only need to solve an ordinary differential equation. To compute (8), we only need to integrate over a small area near point x, especially when the noises are small. In the case of large noises, the multi-grid method ([7]) can be used to achieve linear computational complexity. A simpler alternative is the ADI method, and since the diffusion equation is separable in this case, the ADI method does not even introduce further errors (for solving the discretized system).

To see the difference between the exact solution and the approximate solution (by splitting), we note that

$$T_t^d T_t^c\varphi(x) = I\!E\big[T_t^c\varphi(\zeta(t,x))\big] = I\!E\Big[\varphi\big(\eta(t,\zeta(t,x))\big)\exp\Big\{-\int_0^t(\nabla\cdot b)(\eta(s,\zeta(t,x)))ds\Big\}\Big],$$

$$T_{\frac{t}{2}}^c T_t^d T_{\frac{t}{2}}^c\varphi(x) = T_t^d T_{\frac{t}{2}}^c\varphi(\eta(\frac{t}{2},x))\exp\Big\{-\int_0^{\frac{t}{2}}(\nabla\cdot b)(\eta(s,x))ds\Big\}$$

$$= I\!E\Big[\varphi\Big(\eta(\frac{t}{2},\zeta(t,\eta(\frac{t}{2},x)))\Big)\exp\Big\{-\int_0^{\frac{t}{2}}(\nabla\cdot b)\Big(\eta(s,\zeta(t,\eta(\frac{t}{2},x)))\Big)ds\Big\}\Big]\exp\Big\{-\int_0^{\frac{t}{2}}(\nabla\cdot b)(\eta(s,x))ds\Big\}$$

$$= I\!E\Big[\varphi\Big(\eta(\frac{t}{2},\zeta(t,\eta(\frac{t}{2},x)))\Big)\exp\Big\{-\int_0^{\frac{t}{2}}(\nabla\cdot b)(\eta(s,x))ds - \int_{\frac{t}{2}}^t(\nabla\cdot b)\Big(\eta(s;\frac{t}{2},\zeta(t,\eta(\frac{t}{2},x)))\Big)ds\Big\}\Big],$$

where for obvious reasons we have written $\eta^x(t) = \eta(t,x)$, $\zeta^x(t) = \zeta(t,x)$, and $\eta^{t',x}(t) = \eta(t;t',x)$, the last being the process $\eta(t)$ starting (with probability 1) from the point x at

time t'. Comparing these results with (6), we find that in effect our approximation is to split process $\xi(t)$ into two simpler processes: a deterministic process $\eta(t)$ and (up to a constant) a Wiener process $\zeta(t)$.

In general, if the covariance matrix $\sigma = \sigma(x)$ is not constant but is still assumed to be diagonal to simplify expressions, i.e., $\sigma_{\mu\nu}(x) = \delta_{\mu\nu}a_\nu(x)$, we may rewrite the Fokker-Planck equation in the following form

$$\begin{aligned}\frac{\partial u(t,x)}{\partial t} &= \frac{1}{2}\sum_{\nu=1}^{n}\frac{\partial}{\partial x_\nu}\Big(a_\nu^2(x)\frac{\partial}{\partial x_\nu}u(t,x)\Big)\\ &\quad+\sum_{\nu=1}^{n}\frac{\partial}{\partial x_\nu}\Big((\tilde{a}_\nu(x)-b_\nu(x))u(t,x)\Big),\end{aligned}$$

where $\tilde{a}_\nu(x) = a_\nu(x)\frac{\partial a_\nu}{\partial x_\nu}(x)$ $(\nu = 1,\cdots,n)$. And then split it into two equations

$$\frac{\partial u(t,x)}{\partial t} = \sum_{\nu=1}^{n}\frac{\partial}{\partial x_\nu}\Big((\tilde{a}_\nu(x)-b_\nu(x))u(t,x)\Big),$$

and

$$\frac{\partial u(t,x)}{\partial t} = \frac{1}{2}\sum_{\nu=1}^{n}\frac{\partial}{\partial x_\nu}\Big(a_\nu^2(x)\frac{\partial}{\partial x_\nu}u(t,x)\Big),$$

both of which can be solved in linear computational complexity.

In this general situation, the three solution operators T_t, T_t^c, and T_t^d can also be expressed via certain stochastic (or even deterministic) processes. Indeed, similar to formulas (6), (7), and (8), we have

$$T_t\varphi(x) = I\!E\Big[\varphi(\xi^x(t))\exp\Big\{\int_0^t \nabla\cdot(\tilde{a}-b)(\xi^x(s))ds\Big\}\Big],$$

$$T_t^c\varphi(x) = \varphi(\eta^x(t))\exp\Big\{\int_0^t \nabla\cdot(\tilde{a}-b)(\eta^x(s))ds\Big\},$$

and

$$T_t^d\varphi(x) = I\!E[\varphi(\zeta^x(t))]\,,$$

where processes $\xi^x(t)$, $\eta^x(t)$, and $\zeta^x(t)$ satisfy

$$d\xi^x(t) = (2\tilde{a}(\xi^x(t)) - b(\xi^x(t)))dt + \mathrm{diag}(a_\nu(\xi^x(t)))dW(t),\quad \xi^x(0) = x,$$

$$d\eta^x(t) = (\tilde{a}(\eta^x(t)) - b(\eta^x(t)))dt,\quad \eta^x(0) = x,$$

and

$$d\zeta^x(t) = \tilde{a}(\zeta^x(t))dt + \mathrm{diag}(a_\nu(\zeta^x(t)))dW(t),\quad \zeta^x(0) = x,$$

respectively.

The algorithm by splitting the convection (or drift) term from the (pure) diffusion term as discussed above has a further advantage that much of the computation in (7) and (8) or their generalizations can be performed before the observations are available. This pre-calculation can save the on-line cost and further speed up the real time performance.

4 Adaptive Domain Updating

Even though we have developed optimal solvers for the Fokker-Planck equation, still the nonlinear filtering problem can hardly be solved in real time (especially for large state dimensions) before we can narrow down the size of the domain in which the Fokker-Planck equation is solved. In this section, we present a windowing technique which is based on our convection-diffusion splitting framework.

Assume we are given an initial set D_0 of "important" points which are chosen according to the initial filtering density p_0:

$$D_0 = \{x^1(0), \cdots, x^N(0)\}.$$

Usually these are the points with the largest values of $p_0(x)$ (or with the most important information of $p_0(x)$), Now we describe how to efficiently compute the "important" points in all the subsequent time steps and also the corresponding values of the unnormalized filtering densities at those points.

Let L be a positive integer. We will first construct an enlarged set of LN candidates for the important points and then choose from them the "best" N points according to the observational probability α_k.

For simplicity, we assume again that the covariance matrix σ is constant and diagonal: $\sigma_{\mu\nu}(x) = \delta_{\mu\nu}a_\nu$. Our algorithm can be described as follows.

- Start with $D_0 = \{x^1(0), \cdots, x^N(0)\}$ and $p_0^i = p_0(x_0^i)(1 \le i \le N)$.
- For $k = 1, 2, \cdots$, assuming $D_{k-1} = \{x^1(k-1), \cdots, x^N(k-1)\}$ and $p_{k-1}^i (1 \le i \le N)$ have been computed,

  (1) for $i = 1, \cdots, N$, solve the stochastic integral (or differential) equation

  $$X(t) = x^i(k-1) + \int_0^t b(X(s))ds + \int_0^t \sigma dW(s), \quad t \in (0, \Delta],$$

  with L different sample paths of the Wiener process $W(t)$, including the trivial case $W(t) \equiv 0$, and denote by $\bar{x}^{i,j}$ the solution of the above equation at time $t = \Delta$ with the j-th sample path, $j = 1, \cdots, L$;

  (2) determine the set D_k of N important points $\{x^1(k), \cdots, x^N(k)\}$ according to one of the following two criteria: they are either the points with the N largest values of $\alpha_k(\bar{x}^{i,j})$ for all i and j, or, for each i, $x^i(k)$ has the largest value of $\alpha_k(\bar{x}^{i,j})$ for $j = 1, \cdots, L$;

  (3) for $i = 1, \cdots, N$, compute

  $$p_k^i = \alpha_k(x^i(k)) \sum_{j \in J_k^i} \beta_k^{i,j} p_{k-1}^j,$$

  where $\beta_k^{i,j}$ is computed according to one of the following two formulae:

  $$\beta_k^{i,j} = \exp\Big\{ - \sum_{\nu=1}^{n} \frac{(x_\nu^i(k) - x_\nu^j(k-1) - b_\nu(x^j(k-1))\Delta)^2}{2a_\nu^2 \Delta} \Big\}, \quad \text{or}$$

$$\beta_k^{i,j} = \exp\Big\{ -\sum_{\nu=1}^{n} \frac{(x_\nu^i(k) - x_\nu^j(k-1))^2}{2a_\nu^2\Delta} - (\nabla\cdot b)\Big(\frac{x^i(k)+x^j(k-1)}{2}\Big)\Big\},$$

and $J_k^i = \{j : 1 \le j \le N, \min(\beta_k^{i,j}, p_{k-1}^j) > \tau\}$, τ being a thresholding tolerance.

Since the domain of interest is adaptively moving, the number N of spatial points in the domain can be reduced to a relatively small number, even in three or higher (n) dimensions. In general the number of spatial points is exponential in n, but we are reducing this number for each of the n dimensions. Therefore the computational cost is exponentially reduced in our algorithm.

5 Brief Description of the Subroutines

The complete software will contain two parts: one using an explicit scheme, as described above, and the other using an implicit scheme, which will be addressed later. In the following we explain the subroutines of the explicit part. Corresponding source code is attached at the end of the report.

advection.m
— a function for solving a system of ordinary differntial equations

cutsmall.m
— a function to cut small numbers for thresholding purpose

func_b.m
— a function for the drift term $b(x)$

func_db.m
— a function for the gradient $\nabla \cdot b(x)$

func_h.m
— a function for the observation function $h(x)$

func_p0.m
— a function for the initial filtering density $p_0(x)$

obs_cor.m
— a function for the observational correction $\alpha_k(x)$

generate.m
— a program to generate the (supposedly unknown) target trajectory $X(t)$ and the corresponding observations $z(k)$

initialize.m
— a program to provide initial data and also to initialize global parameters

onestep.m
— a program to compute the filtering density $p_k(x)$

peak.m
— a function for locating the peak of the filtering density

plotting.m
— a function for visualizing the tracking process

prior.m
— a function for computing the prior (transitional) density

run.m
— the main program

6 Numerical Examples

To illustrate the performance of the above algorithm, let us consider two practical examples. Here we take $L = 1$ in the above algorithm.

The first example is a problem of estimating the altitude, velocity, and constant ballistic coefficient of a vertically falling body, given measurements taken at discrete instants of time by a radar that measures range (only) in the presence of white Gaussian noise:

$$\begin{aligned} \text{Signal:} \quad & \dot{X}_1(t) = -X_2(t) - X_1(t)) + a_1\dot{W}_1(t), \\ & \dot{X}_2(t) = -X_2^2(t)X_3(t)\exp\{-\gamma X_1(t)\} + a_2\dot{W}_2(t), \\ & \dot{X}_3(t) = 0 + a_3\dot{W}_3(t), \\ \text{Observation:} \quad & z(k) = \sqrt{M^2 + [X_1(t_k) - H]^2} + v(k), \end{aligned}$$

where $\gamma = 5 \cdot 10^{-5}$, $H = 10^5$, $M = 10^5$, $a_1 = 450$, $a_2 = 88$, $a_3 = 5.6 \cdot 10^{-6}$, $t_k = k\Delta$, $\Delta = 0.5$, $v(k) \sim N(0, (8 \cdot 10^3)^2)$, and the initial target state $(X_1(0), X_2(0), X_3(0))$ has joint density $\pi_0(x_1, x_2, x_3) = \frac{1}{c_0}\exp(-10^{-9}(x_1 - 3 \cdot 10^5)^2 - 2.5 \cdot 10^{-7}(x_2 - 2 \cdot 10^4)^2 - 10^4(x_3 - 3 \cdot 10^{-5})^2)$, c_0 being the normalizing constant.

In our experiments, we have taken $T = 30.0$, $(X_1(0), X_2(0), X_3(0)) = (3 \cdot 10^5, 2 \cdot 10^4, 10^{-3})$, $x_1^i(0) = 2.1 \cdot 10^5 + 3 \cdot 10^4 i$ $(0 \le i \le 10)$, $x_2^j(0) = 1.84 \cdot 10^4 + 2 \cdot 10^3 j$ $(0 \le j \le 10)$, $x_3^k(0) = 3 \cdot 10^{-5} + 1.4 \cdot 10^{-4} k$ $(0 \le k \le 7)$, and so $N = 10 \times 10 \times 7 = 700$. Results of typical tracking steps are shown in the figures at the end of this report.

Our second example is the problem of tracking a noisy Lorenz system with angle-only observations:

$$\begin{aligned} \text{Signal:} \quad & \dot{X}_1(t) = r_1(X_2(t) - X_1(t)) + a_1\dot{W}_1(t), \\ & \dot{X}_2(t) = r_2X_1(t) - X_1(t)X_3(t) - X_2(t) + a_2\dot{W}_2(t), \\ & \dot{X}_3(t) = X_1(t)X_2(t) - r_3X_3(t) + a_3\dot{W}_3(t), \\ \text{Observation:} \quad & z_1(k) = \text{sign}(X_2(t_k))\arccos\frac{X_1(t_k)}{\sqrt{X_1^2(t_k)+X_2^2(t_k)}} + v_1(k), \\ & z_2(k) = \arcsin\frac{X_3(t_k)}{\sqrt{X_1^2(t_k)+X_2^2(t_k)+X_3^2(t_k)}} + v_2(k), \end{aligned}$$

where $r_1 = 10$, $r_2 = 28$, $r_3 = 8/3$, $a_1 = 0.045$, $a_2 = 0.023$, $a_3 = 0.012$, $t_k = k\Delta$, $\Delta = \frac{1}{32}$, $v_1(k) \sim N(0, .64^2)$, $v_2(k) \sim N(0, .36^2)$, and the initial target state $(X_1(0), X_2(0), X_3(0))$ has joint density $\pi_0(x_1, x_2, x_3) = \frac{1}{c_0}\exp(-25(x_1 - 4.5)^2 - 16(x_2 + 4.5)^2 - 9(x_3 - 19)^2)$, c_0 being the normalizing constant.

In our experiments, we took $T = 4.0$, $(X_1(0), X_2(0), X_3(0)) = (5, -5, 20)$, $x_1^i(0) = 4 + \frac{i}{5}$ $(0 \le i \le 10)$, $x_2^j(0) = -6 + \frac{j}{5}$ $(0 \le j \le 10)$, $x_3^k(0) = 19 + \frac{k}{5}$ $(0 \le k \le 10)$, and so

$N = 10 \times 10 \times 10 = 1000$. Results of typical tracking steps are shown in the figures at the end of this report.

The computational cost for these two examples (in terms of CPU seconds and FLOPS per time step) are given in Table 1. (Note: A portion of CPU time for the Lorenz example was used for better viualization.)

Table 1: Computational complexity

Problem	N	cpu	flops
Ballistic	700	0.065	80659
Lorenz	1000	0.67	135406

In conclusion, the filtering algorithm proposed in this report is based on the exact optimal filter and so applies to systems with high nonlinearity and different noise levels. On the other hand, the new algorithm is fast, in that the involved calculations have linear comlexity per time step and the number of points at which the calculations are performed is small. In other words, we have developed a fast optimal nonlinear filter.

Appendix

MATLAB source code will be included at the end of this report.

References

[1] V. J. Aidala, Kalman filter behavior in bearings-only tracking applications, *IEEE Trans. on Aerospace and Electronic Systems*, Vol. AES-15, No.1, January 1979, pp. 29-39.

[2] Y. Bar-Shalom and X.-R. Li, *Estimation and Tracking: Principles, Techniques, and Software.* Boston: Artech House, Inc., 1993.

[3] W. D. Blair and G. A. Watson, IMM algorithm for solution to benchmark problem for tracking maneuvering targets. *Proceedings of SPIE Acquisition, Tracking, and Pointing VIII* (Orlando, FL.), SPIE, 1994, pp. 476-488.

[4] F. El-Hawary and Y. Jing, Robust regression-based EKF for tracking underwater targets, *IEEE J. of Oceanic Engineering*, Vol. 20, No.1 1995, pp. 31-41.

[5] B. Etkin. *Dynamics of Atmospheric Flight.* New York: John Wiley & Sons, Inc., 1972.

[6] B. Gustafsson, H.-O. Kreiss, and J. Oliger. *Time Dependent Problems and Difference Methods.* John Wiley & Sons, Inc., New York, 1995.

[7] W. Hackbusch. *Multi-grid Methods and Applications.* Springer-Verlag, Berlin, 1985.

[8] Z. S. Haddard and S. R. Simanca. Filtering image records using wavelets and the Zakai equation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 17, No. 11, Nov. 1995, pp. 1069-1078.

[9] H. D. Hoelzer, G. W. Jonson, and A. O. Cohen, Modified Polar Coordinates – The Key to Well Behaved Bearings Only Ranging, IR&D Report 78-M19-0001A, IBM Federal Systems Division, Shipboard and Defense Systems, Manassas VA 22110, August 31, 1978.

[10] A. H. Jazwinski. *Stochastic Processes and Filtering Theory.* Academic Press, New York, 1970.

[11] L. Kang et al. *Decomposition Methods for Numerically Solving Higher Dimensional Partial Differential Equations.* Shanghai Science and Technology Press, Shanghai, 1990.

[12] S. V. Lototsky, R. Mikulevicius, B. L. Rozovksii. Nonlinear filtering revisited: a spectral approach. *SIAM J. Control Optim.*, Vol. 35, No.2, March 1997, pp. 435-461.

[13] S. V. Lototsky and B. L. Rozovskii. Recursive nonlinear filter for a continuous-discrete time model: separation of parameters and observations. *IEEE Transactions on Automatic Control*, 1997. (To appear)

[14] S. V. Lototsky, C. Rao, B. L. Rozovskii. Fast nonlinear filter for continuous-discrete time multiple models. *Proceedings of the 35th Conference on Decision and Control*, Kobe, Japan, 1996, Omnipress, Madison, Wisconsin, Vol. 4, pp. 4060-4064.

[15] G. I. Marchuk. Splitting and alternating direction methods. In: *Handbook of Numerical Analysis, Vol. I*, Ph. G. Ciarlet and J.-L. Lions (eds.), North-Holland, Amsterdam, 1990, pp. 197-462.

[16] G. N. Milstein, *Numerical Integration of Stochastic Differential Equations.* Dordrecht: Kluwer Academic Publishers, 1995.

[17] C. Rao and B. L. Rozovskii. A fast filter for nonlinear systems with discrete observations. *Proceedings of the Second IASC World Conference*, Pasadena, February 1997.

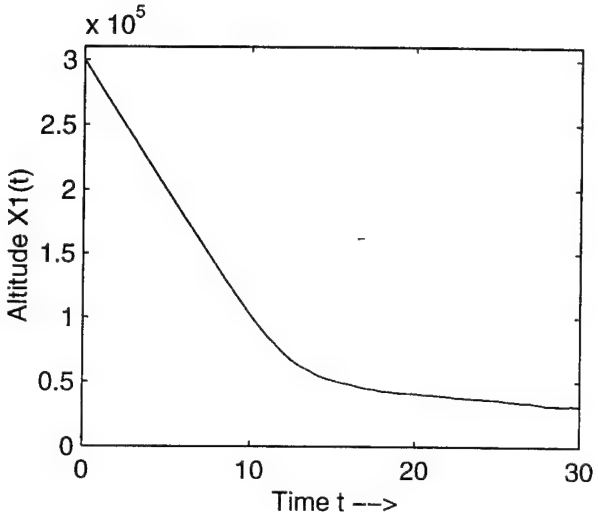
[18] C. Rao. Mathematical models of maneuvering targets. Technical Report, University of Southern California, 1996.

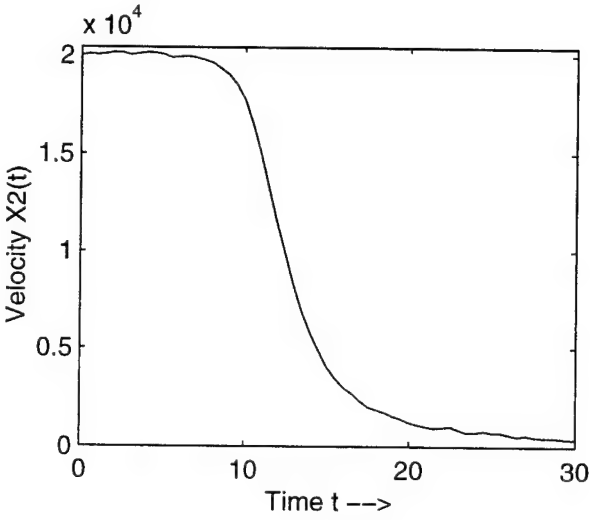
[19] C. Rao. A new difference scheme and related splitting methods for convection-diffusion equations. Technical Report, University of Southern California, 1997.

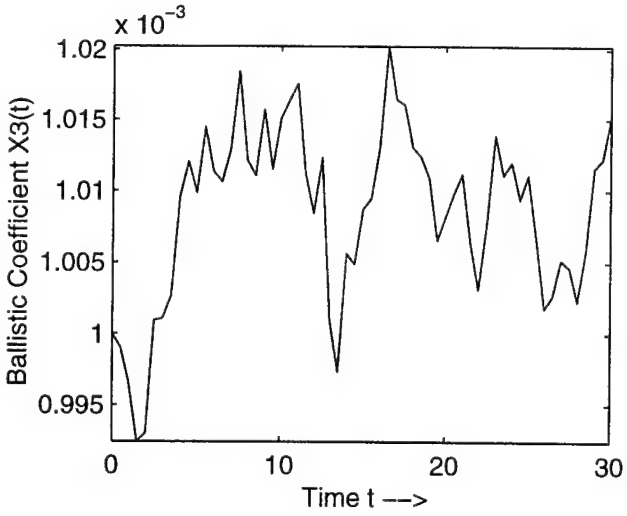
[20] D. D. Sworder, P. F. Singer, D. Doria, and R. G. Hutchins, Image-enhanced estimation methods. *Proceedings of the IEEE*, Vol. 81 (1993), No. 6, pp. 797-812.

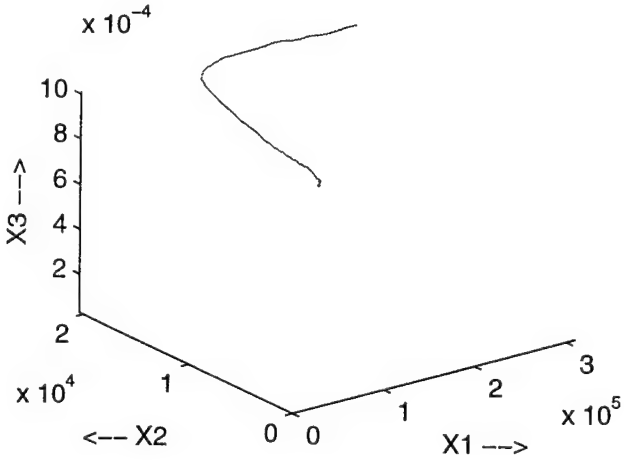
[21] N. N. Yanenko. *The Method of Fractional Steps: The Solution of Problems of Mathematical Physics in Several Variables.* Springer-Verlag, Berlin, 1971.

The ballistic example.

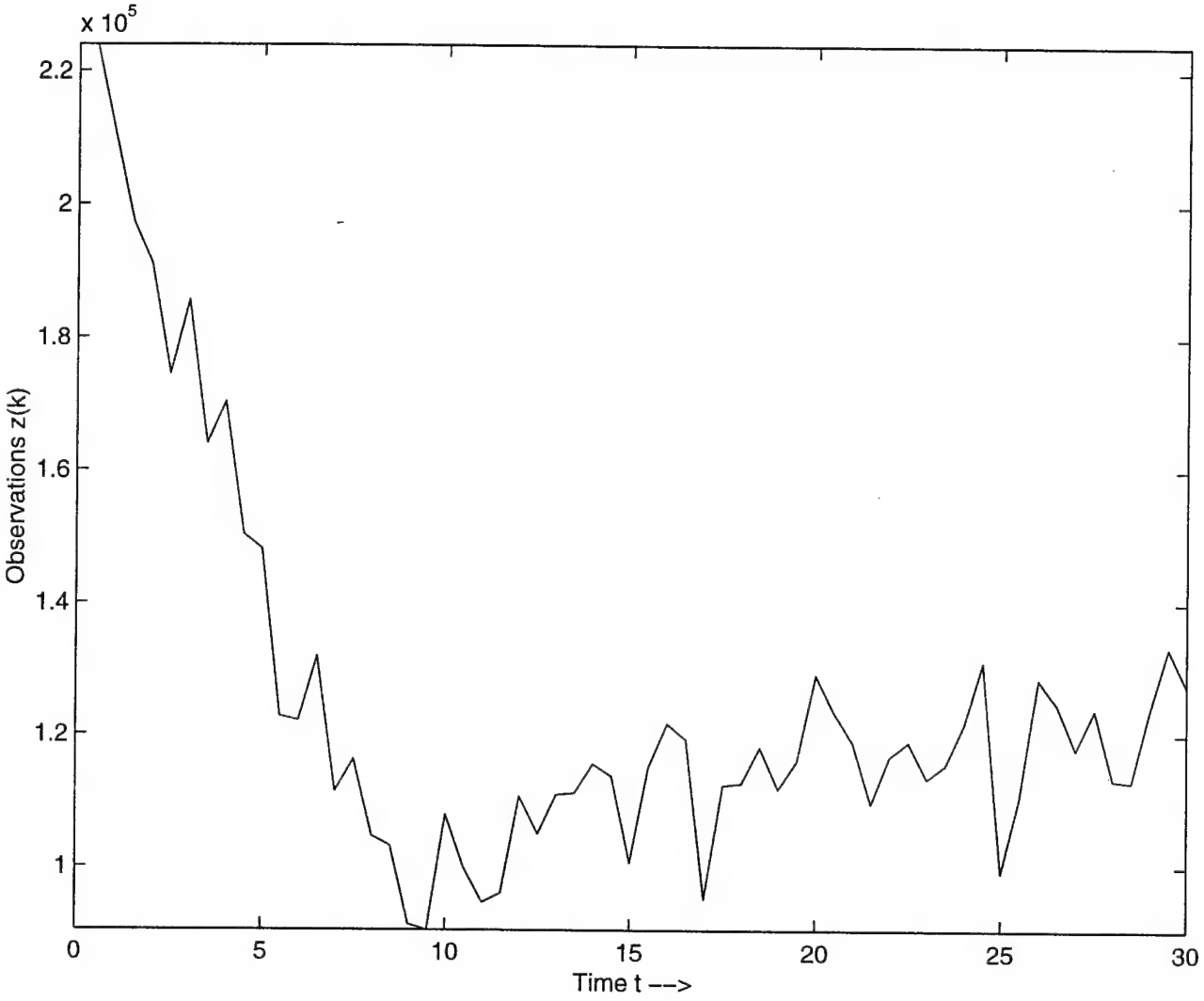




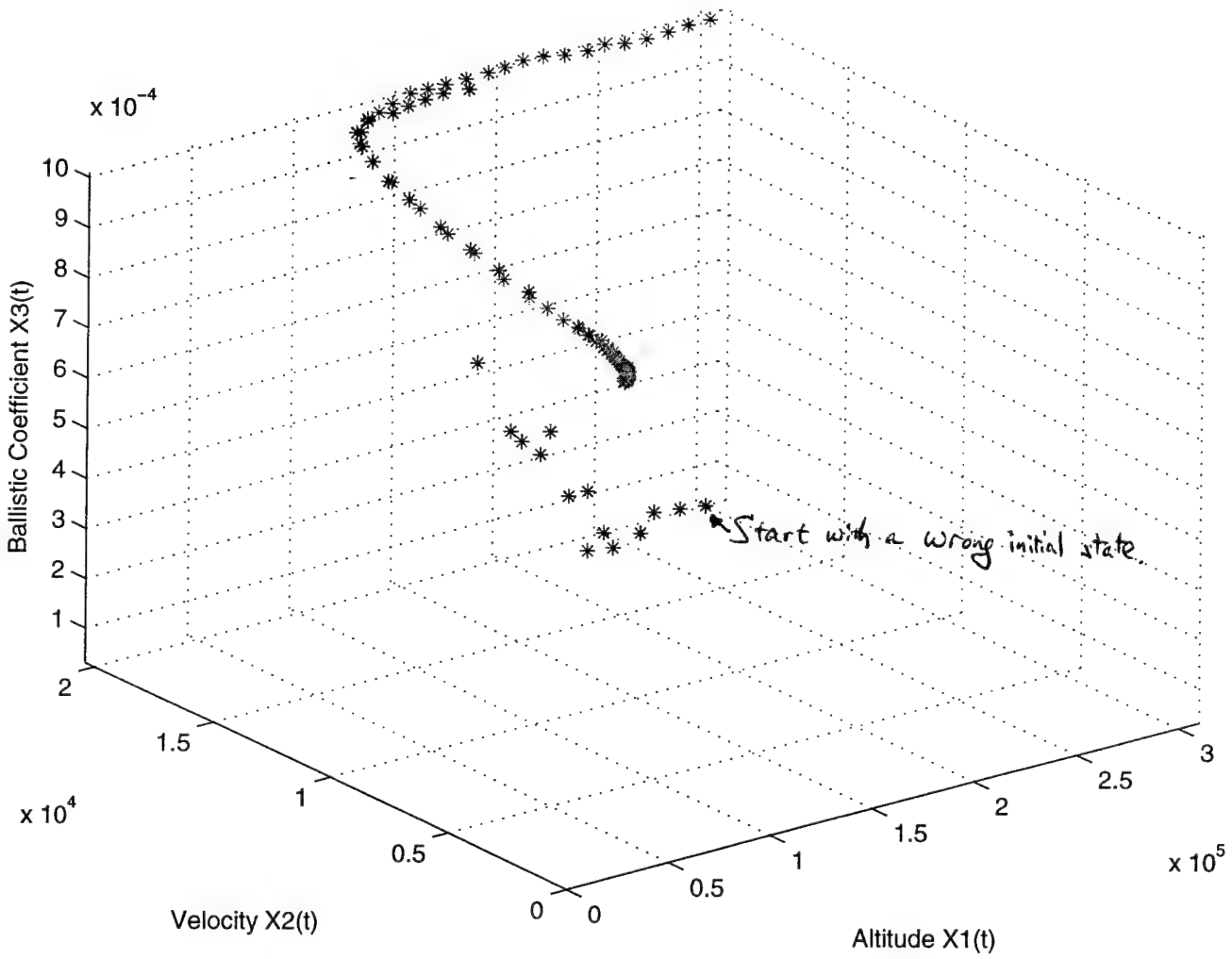




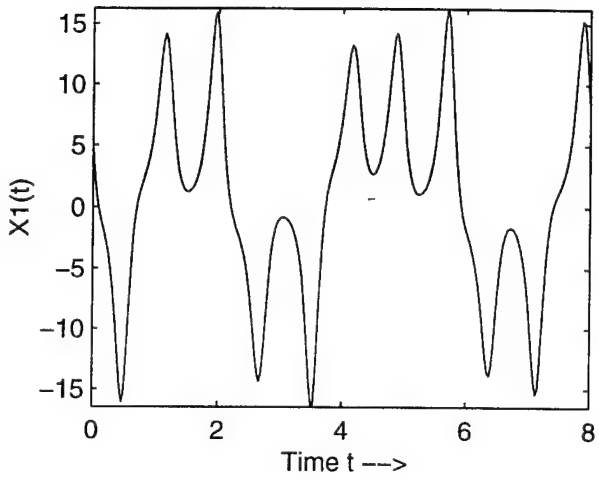
Range Measurements

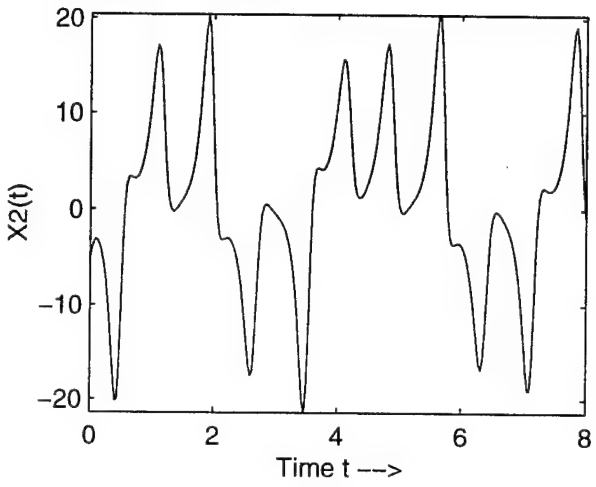


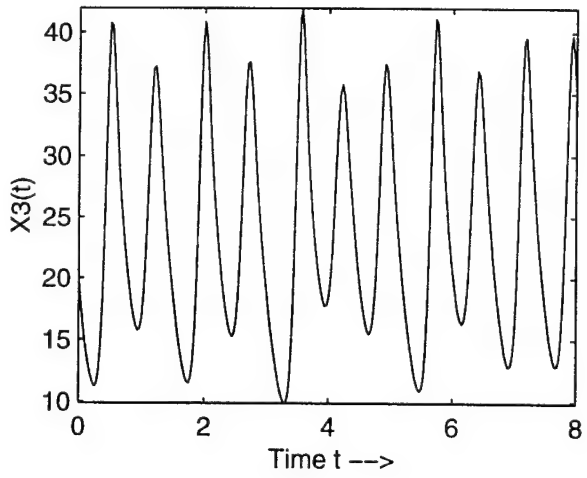
Tracking process in phase space

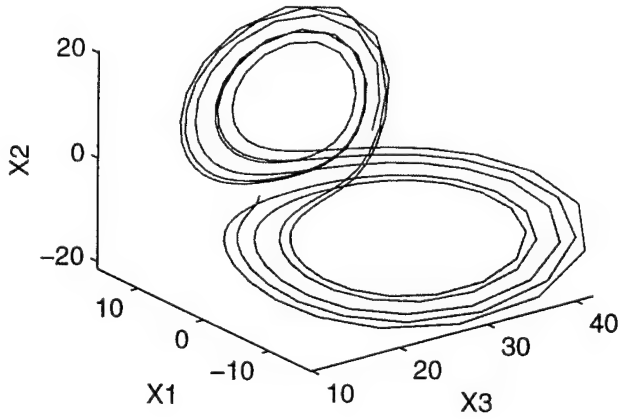


The Lorenz system:

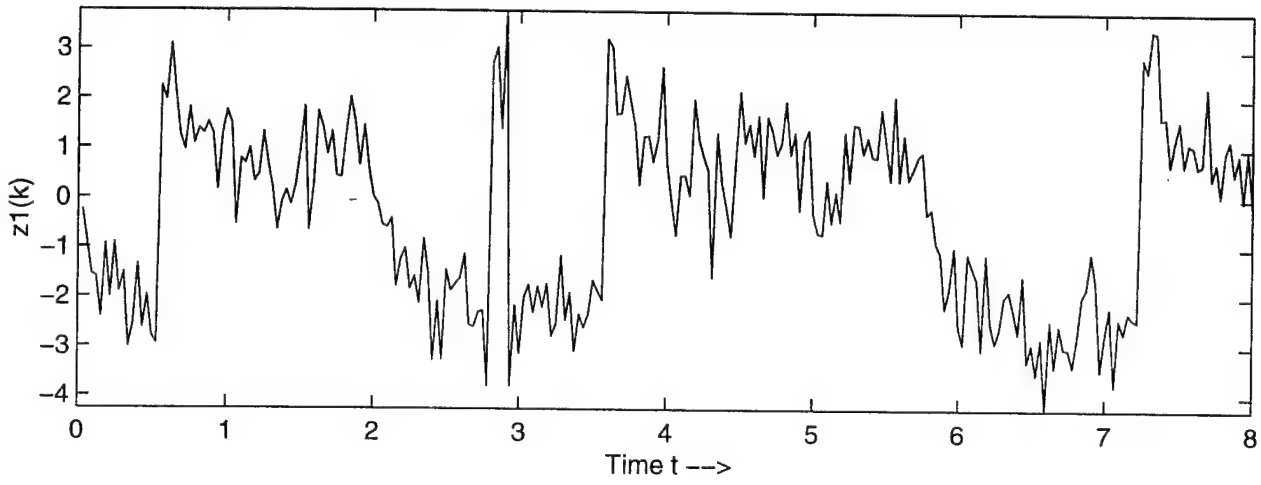


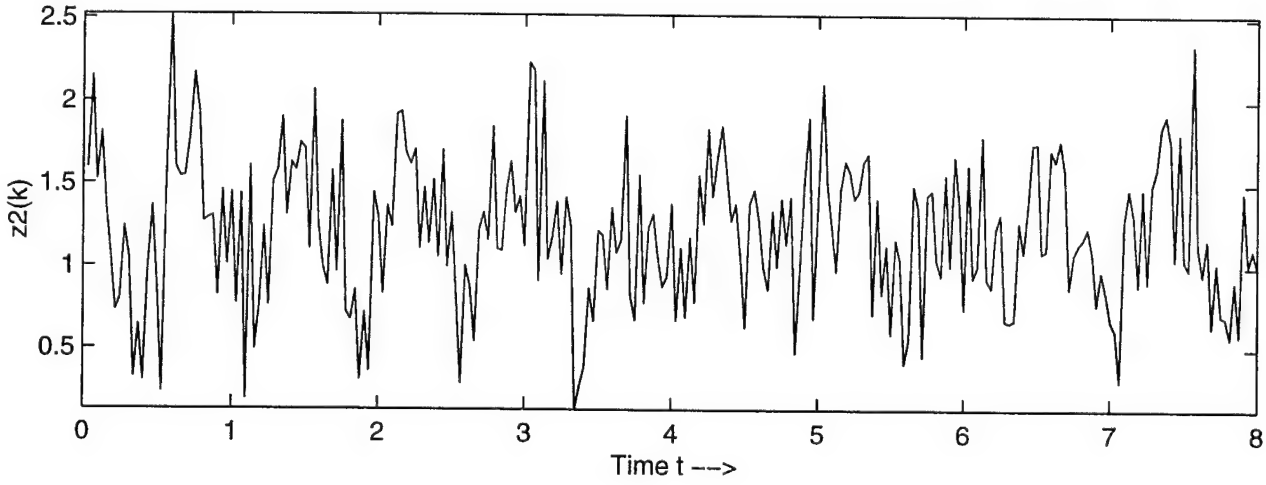


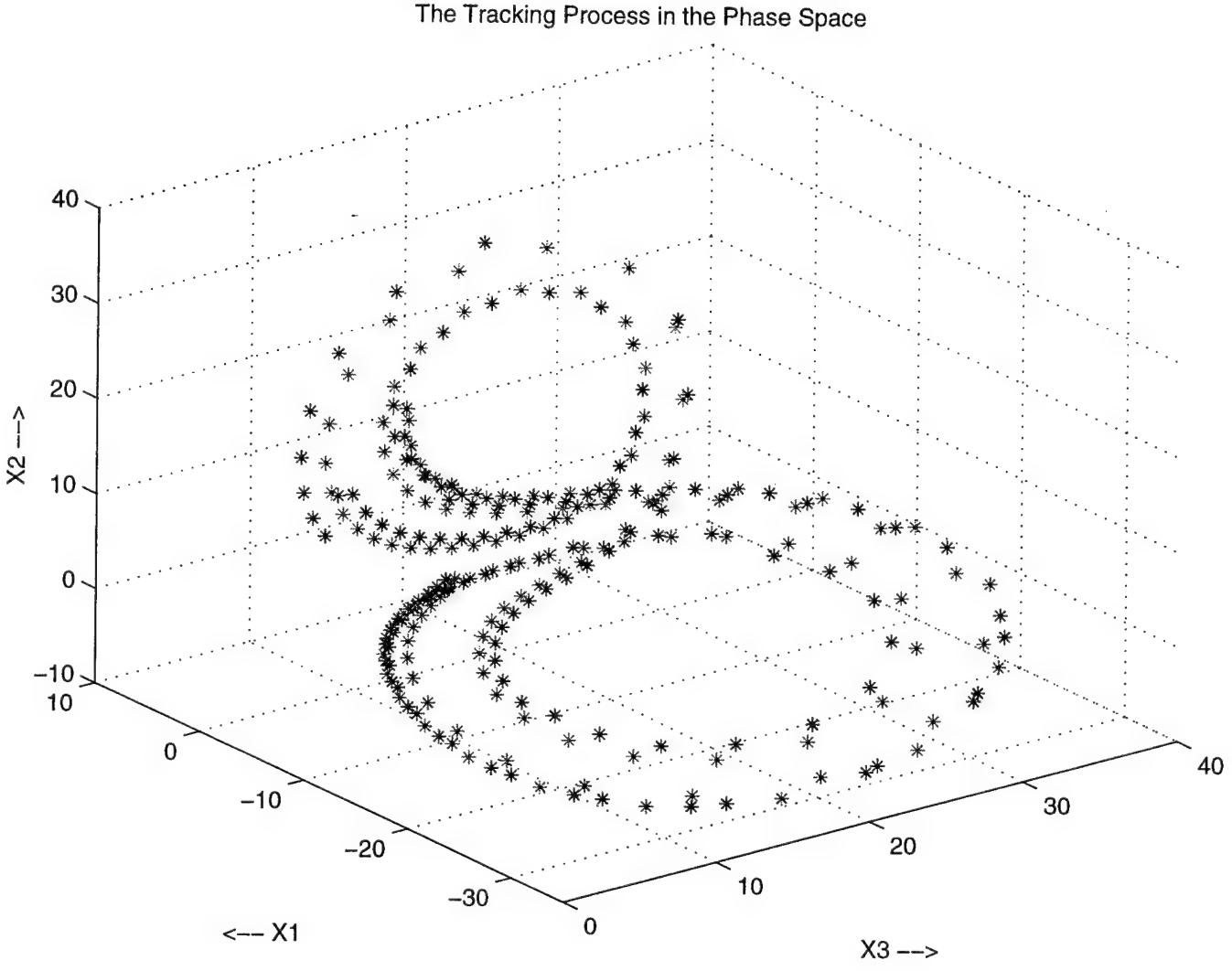




Angle-only Observations






The Tracking Process in the Phase Space
40
30
20
10
0
−10
X2 −−>
10
0
−10
−20
−30
<−− X1
0
10
20
30
40
X3 −−>

%%%%%%%%%%%
1. Ballistic

cpu1 = 0.0645
flops1 = 80659
cpu60 = 3.88
flops60 = 4839540

gam = 5e-5;
H = 1e5; %0; % 2e5;
M = 1e5;

Tfinal = 30.0;
Kfinal = 60; %30;
Kmax = 60; %30

nx = 10;
ny = 10;
nz = 7;
xa = 2.1e5; xb = 3.1e5;
ya = 1.84e4; yb = 2.04e4;
za = 3e-5; zb = 1.01e-3;

sigmaa = [4.5e2, 8.8e1, 5.6e-6]; %1e-6; %1e-4;
delta = 0.8e4;

m0 = [3e5, 2e4, 3e-5];
var0_inv = [1e-9, 2.5e-7, 1e4];
%var0_inv = [1e-6, 2.5e-7, 1e4];

X0 = [3e5, 2e4, 1e-3];
Z0 = 0;

%%%%%%%%%
2. Lorenz

cpu1 = 0.67
flops1 = 135406
cpu128 = 86.08
flops128 = 17331968

ss = 10;
aa = 28;
bb = 8/3;

Tfinal = 8.0;
Kfinal = 256;
Kmax = 128; %64; %256;

nx = 10;
ny = 10;

```
nz = 10;
xa = 4;  xb = 6;
ya = -4;  yb = -6;
za = 19;  zb = 21;

sigmaa = [0.045, 0.023, 0.012];
delta = [0.64, 0.36];

m0 = [4.5, -4.5, 19];
var0_inv = [25, 16, 9];

X0 = [5, -5, 20];
Z0 = [0,0];


%%%%%%%%%%
3. Sigma3D

cpu1 =    0.65
flops1 =      115394
cpu150 = 97.75
flops150 =    17309100

Tfinal = 2.0;
Kfinal = 256; Kt = Kfinal;
Kmax = 150 %128; %64; %256;

nx = 12;
ny = 10;
nz = 8;
xa = 0.15;  xb = 0.75;
ya = 0.30;  yb = 0.60;
za = 0.14;  zb = 0.30;

sigmaa = [0.045, 0.023, 0.012];
%delta = [0.24, 0.12];
delta = [0.64, 0.36];

m0 = [0.45, 0.42, 0.20];
var0_inv = [100, 121, 64];
%var0_inv = [82, 100, 64];

%X0 = [0.75, -0.5, -0.25];
X0 = [3.6/4/2, 1./2, 1./4];
Z0 = [0,0];
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      New operator splitting method for solving
%%      dX = b(X)dt + sigmaa(X)dW,  X(0) = m0
%%      z(k) = h(X(t_k)) + delta(k)v(k)
%%      Chuanxia Rao
%%      January 1996
%%      January 1997
%%      May 1997
%%      Sep 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      This is the main program.
%%-------------------------------------
%%      Dependencies:
%%      initialize.m
%%      moreinit.m
%%      generate.m
%%      onestep.m
%%      plotting.m
%%-------------------------------------
%%      Output:
%%      graphs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear;

initialize;             %% initial data and parameters
generate;               %% to generate processes X and Z

figure; axHndl=gca;
set(axHndl, ...
        'XLim',[0 40],'YLim',[-35 10],'ZLim',[-10 40], ...
        'Drawmode', 'fast', ...
        'Visible','on', ...
        'NextPlot','add', ...
        'View',[-37.5,30]);
xlabel('Z');
ylabel('X');
zlabel('Y');
    head0 = line( ...
        'color','g', ...
        'Marker','.', ...
        'markersize',25, ...
        'erase','xor', ...
        'xdata',X(1,3),'ydata',X(1,1),'zdata',X(1,2));
    tail0 = line( ...
        'color','b', ...
        'LineStyle','-', ...
        'erase','none', ...
        'xdata',[],'ydata',[],'zdata',[]);
    head1 = line( ...
        'color','r', ...
        'Marker','.', ...
        'markersize',25, ...
        'erase','xor', ...
        'xdata',m0(3),'ydata',m0(1),'zdata',m0(2));
    tail1 = line( ...
        'color','y', ...
        'LineStyle','-', ...
        'erase','none', ...
        'xdata',[],'ydata',[],'zdata',[]);

%       xyz = X(1,:);
```

```
%       plotting(xyz, pk);
%       hold on;

        Y0 = X(1,:)' * ones(1,2);
        Y1 = m0(:) * ones(1,2);

kstart=1;       kstop=Kmax;
%kstart=Kmax+1; kstop=Kfinal-1;
%%global time;

        count = flops;
        cpu = cputime;
for ktime = kstart:kstop,

        k1 = ktime + 1;
%%      time = t0 + dt*ktime;
        fprintf(1, '%c', '  step ');
        fprintf(1, '%d\n', ktime);

        onestep;

kplot = ktime/plot_step;
if ktime==1 | kplot == fix(kplot),
        xyz = X(k1,:)';
        Y0 = [xyz Y0(:,1)];
        set(head0,'xdata',Y0(3,1),'ydata',Y0(1,1),'zdata',Y0(2,1))
        set(tail0,'xdata',Y0(3,1:2),'ydata',Y0(1,1:2),'zdata',Y0(2,1:2))
        drawnow;
        xyz = peak(pk);
        Y1 = [xyz Y1(:,1:2)];
        set(head1,'xdata',Y1(3,1),'ydata',Y1(1,1),'zdata',Y1(2,1))
        set(tail1,'xdata',Y1(3,1:2),'ydata',Y1(1,1:2),'zdata',Y1(2,1:2))
        drawnow;
%       plotting(xyz, pk);
end

end
%       hold off;
        count = flops - count;
        cpu = cputime - cpu;
        cpu_on1 = cpu / (kstop-kstart+1)
        flops_on1 = count / (kstop-kstart+1)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      New operator splitting method for solving
%%      dX = b(X)dt + sigmaa(X)dW,  X(0) = m0
%%      z(k) = h(X(t_k)) + delta(k)v(k)
%%      Chuanxia Rao
%%      January 1996
%%      January 1997
%%      September 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      This is the main program.
%%-------------------------------------
%%      Dependencies:
%%      initialize.m
%%      moreinit.m
%%      generate.m
%%      onestep.m
%%      plotting.m
%%-------------------------------------
%%      Output:
%%      graphs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear;

initialize;             %% initial data and parameters
generate;               %% to generate processes X and Z

figure;
        xyz = X(1,:);
        plotting(xyz, pk);
kstart=1;       kstop=Kmax;
%kstart=Kmax+1; kstop=Kfinal-1;

        count = flops;
        cpu = cputime;
%%global time;
for ktime = kstart:kstop,

        k1 = ktime + 1;
%%      time = t0 + dt*ktime;
        fprintf(1, '%c', '  step ');
        fprintf(1, '%d\n', ktime);

        onestep;

kplot = ktime/plot_step;
if ktime==1 | kplot == fix(kplot),
        xyz = X(k1,:);
        plotting(xyz, pk);
end

end
        count = flops - count;
        cpu = cputime - cpu;
        cpu_on1 = cpu / (kstop-kstart+1)
        flops_on1 = count / (kstop-kstart+1)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Initial data for the cont.-discr. model:
%%      dX = b(X)dt + sigmaa*dW
%%      X(0) ~ N(m0,var0)
%%      z(k) = h(X(t_k)) + delta*v(k)
%%----------------------------
%%      Chuanxia Rao
%%      Februay 1996
%%      Februry 1997
%%      July 1997
%%----------------------------
%%      Called in:
%%      all other parts
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global  dim dim_obs
global  nx ny nz
global  xa ya za
global  xb yb zb
global  xx yy zz
global  xend0 yend0 zend0
global  xend1 yend1 zend1
global  m0 var0_inv
global  gam Msquared H

global  dt dt2
global  denom
global  denom_obs
%global nxp1 nyp1 nzp1

global substeps
global small %large
global resol_vert


        disp(' Initialization ...');
        dim = 3;
        dim_obs = 1;

Tfinal = 30.0;
Kfinal = 60; %30;
Kmax = 60; %30
Kt = Kfinal;

nx = 10;
ny = 10;
nz = 7;
xa = 2.1e5;  xb = 3.1e5;
ya = 1.84e4;  yb = 2.04e4;
za = 3e-5;  zb = 1.01e-3;

xend0 = 0;  xend1 = 3.1e5;
yend0 = 0;  yend1 = 2.04e4;
zend0 = 3e-5;  zend1 = 1.01e-3;
%Nx = [nx,ny,nz];
%Lx = [3,2,1];
%Lx = [1,1,1];                   %% bandwidth of local propogation:
                                 %% It depends on (dx*dx)/(dt*sigmaa^2).

plot_step = 1; %2;               %% plot in every plot_step steps
resol_vert = 0.005;              %% for plotting target position

substeps = 1;
small = 1e-16;                   %% threshold for positive probability
```

```
%large = -log(small);

%sigmaa = [0.0, 0.0, 0.0];      %% coefficients in signal noise
sigmaa = [4.5e2, 8.8e1, 5.6e-6];        %1e-6; %1e-4;
%delta = 0.0;                   %% coefficients in observ. noise
delta = 0.8e4;

m0 = [3e5, 2e4, 3e-5];          %% initial mean
var0_inv = [1e-9, 2.5e-7, 1e4]; %% initial variance, its inverse
%var0_inv = [1e-6, 2.5e-7, 1e4];

X0 = [3e5, 2e4, 1e-3];
Z0 = 0;

gam = 5e-5;
H = 1e5; %0;  % 2e5;
M = 1e5;

%%
%%  Further initialization:
%%  -- no changes needed --
%%

Msquared = M*M;
dt = Tfinal/Kfinal;
dx = (xb-xa)/nx;
dy = (yb-ya)/ny;
dz = (zb-za)/nz;

dt2 = dt/2;
%dt4 = dt/4;
nxp1 = nx + 1;
nyp1 = ny + 1;
nzp1 = nz + 1;

denom_obs = 2 * delta.^2;               %% a row vector, not a diagonal matrix
denom = (2*dt) * sigmaa.^2;             %% a row vector
%const_norm = sigmaa... * sqrt(2*pi*dt)^dim;

disp(' Setting-up of initial density and observation function ...')
        xx = xa + dx * (0:nx);
        yy = ya + dy * (0:ny);
        zz = za + dz * (0:nz);
        pk = func_p0(xx, yy, zz);
%       pk = cutsmall(pk);

        x1d = xx;
        y1d = yy;
        z1d = zz;
        xx = zeros(nxp1,nyp1,nzp1);
        yy = xx;
        zz = yy;
for i=1:nxp1,
for j=1:nyp1,
for k=1:nzp1,
        xx(i,j,k) = x1d(i);
        yy(i,j,k) = y1d(j);
        zz(i,j,k) = z1d(k);
end
end
end

%       [hh1, hh2] = func_h(xx, yy, zz);
%%      hh1 = cutsmall(hh1);
```

```
function  p0 = func_p0(x,y,z)
% usage:  p0 = func_p0(x,y,z)
% where   x,y,z are vectors.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The initial density function
%%      Chuanxia Rao
%%      Februry 1997
%%      May 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Called in:
%%      moreinit.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global m0 var0_inv

%       temp = sqrt(2*pi)^3;
%       temp0 = temp*sqrt(var0_inv(1)*var0_inv(2)*var0_inv(3));
for i=1:length(x),
        temp1 = (x(i)-m0(1))^2 * var0_inv(1);
for j=1:length(y),
        temp2 = (y(j)-m0(2))^2 * var0_inv(2);
for k=1:length(z),
        temp3 = (z(k)-m0(3))^2 * var0_inv(3);
        temp3 = temp1 + temp2 + temp3;
        p0(i,j,k) = exp( -temp3/2 );
end
end
end
%       p0 = p0 / temp0;
        p0 = p0 / max( max( max(p0,[],3),[],2 ),[],1 );
```

```
function  [b1,b2,b3] = func_b(x,y,z)
% usage:  [b1,b2,b3] = func_b(x,y,z)
% where   x,y,z and b1,b2,b3 are of the same size.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The signal propogation function
%%      Chuanxia Rao
%%      Februry 1997
%%      August 1997
%%----------------------------------------
%%      Called in:
%%      generate.m
%%      advection.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global gam

        b1 = -y;
        b2 = -exp(-gam*x) .* y.^2 .* z;
        b3 = zeros(size(y));

%       b = [b1, b2, b3];
```

```
function  result = func_db(x,y,z)
% usage:  result = func_db(x,y,z)
% where   x,y,z, and result are of the same size.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The scalar function  Delta * b
%%      Chuanxia Rao
%%      August 1997
%%      May 1997
%%--------------------------------------
%%      Called in:
%%      prior.m (& onestep.m)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global gam

if (size(x)==size(y) & size(y)==size(z)),
%       db1 = zeros(size(x));
%       db2 = -2 * exp(-gam*x) .* y .* z;
%       db3 = zeros(size(z));
%       db = db1 + db2 + db3;
        result = -2 * exp(-gam*x) .* y .* z;
else
        disp(' Warning: x, y, and z have different sizes ?!')
        result = zeros(size(y));
end

%n1 = length(x);
%n2 = length(y);
%n3 = length(z);
%       db1 = zeros(n1,n2,n3);
%       db2 = zeros(n1,n2,n3);
%       db3 = zeros(n1,n2,n3);
%       db = zeros(n1,n2,n3);
%%      db = db1 + db2 + db3;
%%      db1 = db / 3;
%%      db2 = db1;
%%      db3 = db1;
```

```
function  hh = func_h(x,y,z)
% usage:  hh = func_h(x,y,z)
% where   x,y,z and hh are of the same size.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The observation function
%%      Chuanxia Rao
%%      August 1997
%%      April 1997
%%----------------------------------------
%%      Called in:
%%      obs_cor.m
%%      generate.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global  Msquared H

if (size(x)==size(y) & size(y)==size(z)),
        temp = Msquared + (x - H).^2;
        hh = sqrt(temp);
else
        disp(' Warning: x, y, and z have different sizes ?!')
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Initial data for the cont.-discr. model:
%%      dX = b(X)dt + sigmaa*dW
%%      X(0) ~ N(m0,var0)
%%      z(k) = h(X(t_k)) + delta*v(k)
%%----------------------------
%%      Chuanxia Rao
%%      Februay 1996
%%      Februry 1997
%%      July 1997
%%----------------------------
%%      Called in:
%%      all other parts
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global  dim dim_obs
global  nx ny nz
global  xa ya za
global  xb yb zb
global  xx yy zz
global  xend0 yend0 zend0
global  xend1 yend1 zend1
global  m0 var0_inv

global  dt dt2
global  denom
global  denom_obs
%global nxp1 nyp1 nzp1

global substeps
global small %large
global resol_vert

global  ss aa bb
        %% constants in the Lorenz's system.

ss = 10;
aa = 28;
bb = 8/3;

        disp(' Initialization ...');
        dim = 3;
        dim_obs = 2;

Tfinal = 8.0;
Kfinal = 256; Kt = Kfinal;
Kmax = 128; %64; %256;
%Kmax = Kfinal - 1;
%Kmax = Kfinal / 4;

nx = 10;
ny = 10;
nz = 10;
xa = 4;  xb = 6;
ya = -4;  yb = -6;
za = 19;  zb = 21;
%nx = 50;
%ny = 50;
%nz = 30;
%xa = -0.25;  xb = 0.75;
%ya = -0.40;  yb = 0.60;
%za = 0.0;  zb = 0.30;

xend0 = -35;  xend1 = 10;
```

```
yend0 = -10;  yend1 = 40;
zend0 = 0;    zend1 = 40;

%xa = [-0.95, -0.6, -0.3];
%xb = [0.75, 0.6, 0.3];
%Nx = [nx,ny,nz];
%Lx = [3,2,1];
%Lx = [1,1,1];                  %% bandwidth of local propogation:
                                %% It depends on (dx*dx)/(dt*sigmaa^2).

%sigmaa = [0.0, 0.0, 0.0];      %% coefficients in signal noise
sigmaa = [0.045, 0.023, 0.012];
%delta = [0.0, 0.0];            %% coefficients in observ. noise
delta = [0.64, 0.36];
%delta = [pi, pi/2];

m0 = [4.5, -4.5, 19];           %% initial mean
var0_inv = [25, 16, 9];         %% initial variance, its inverse

%    X0(1)=rand*35-30;
%    X0(2)=rand*40-5;
%    X0(3)=rand*30+5;
X0 = [5, -5, 20];
%X0 = [-5, 20, 5];      %% only one cycle ?!
%X0 = [3.6/4/2, 1./2, 1./4];
Z0 = [0,0];

plot_step = 1; %2;              %% plot in every plot_step steps
resol_vert = 0.01;              %% for plotting target position

substeps = 1;
small = 1e-16;                  %% threshold for positive probability
%large = -log(small);


%%
%%  Further initialization:
%%  -- no changes needed --
%%

dt = Tfinal/Kfinal;
dx = (xb-xa)/nx;
dy = (yb-ya)/ny;
dz = (zb-za)/nz;

dt2 = dt/2;
%dt4 = dt/4;
nxp1 = nx + 1;
nyp1 = ny + 1;
nzp1 = nz + 1;

denom_obs = 2 * delta.^2;               %% a row vector, not a diagonal matrix
denom = (2*dt) * sigmaa.^2;             %% a row vector
%const_norm = sigmaa... * sqrt(2*pi*dt)^dim;

disp(' Setting-up of initial density and observation function ...')
        xx = xa + dx * (0:nx);
        yy = ya + dy * (0:ny);
        zz = za + dz * (0:nz);
        pk = func_p0(xx, yy, zz);
%       pk = cutsmall(pk);

        x1d = xx;
        y1d = yy;
```

```
        z1d = zz;
        xx = zeros(nxp1,nyp1,nzp1);
        yy = xx;
        zz = yy;
for i=1:nxp1,
for j=1:nyp1,
for k=1:nzp1,
        xx(i,j,k) = x1d(i);
        yy(i,j,k) = y1d(j);
        zz(i,j,k) = z1d(k);
end
end
end

%       [hh1, hh2] = func_h(xx, yy, zz);
%%      hh1 = cutsmall(hh1);
%%      hh2 = cutsmall(hh2);
```

```
function  p0 = func_p0(x,y,z)
% usage:  p0 = func_p0(x,y,z)
% where   x,y,z are vectors.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The initial density function
%%      Chuanxia Rao
%%      Februry 1997
%%      May 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Called in:
%%      moreinit.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global m0 var0_inv

%       temp = sqrt(2*pi)^3;
%       temp0 = temp*sqrt(var0_inv(1)*var0_inv(2)*var0_inv(3));
for i=1:length(x),
        temp1 = (x(i)-m0(1))^2 * var0_inv(1);
for j=1:length(y),
        temp2 = (y(j)-m0(2))^2 * var0_inv(2);
for k=1:length(z),
        temp3 = (z(k)-m0(3))^2 * var0_inv(3);
        temp3 = temp1 + temp2 + temp3;
        p0(i,j,k) = exp( -temp3/2 );
end
end
end
%       p0 = p0 / temp0;
        p0 = p0 / max( max( max(p0,[],3),[],2 ),[],1 );
```

```
function  [b1,b2,b3] = func_b(x,y,z)
% usage:  [b1,b2,b3] = func_b(x,y,z)
% where   x,y,z and b1,b2,b3 are of the same size.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The signal propogation function
%%      Chuanxia Rao
%%      Februry 1997
%%      May 1997
%%----------------------------------------
%%      Called in:
%%      generate.m
%%      advection.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global  ss aa bb

        b1 = ss * (y - x);

        b2 = -x.*z + aa*x - y;

        b3 = x.*y - bb*z;

%       b = [b1, b2, b3];
```

```
function  result = func_db(x,y,z)
% usage:  result = func_db(x,y,z)
% where   x,y,z, and result are of the same size.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The scalar function  Delta * b
%%      Chuanxia Rao
%%      July 1997
%%      May 1997
%%----------------------------------------
%%      Called in:
%%      prior.m (& onestep.m)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global  ss bb

%if (size(x)==size(y) & size(y)==size(z)),
        db1 = -ss * ones(size(x));
        db2 = - ones(size(y));
        db3 = -bb * ones(size(z));
        result = db1 + db2 + db3;
%else
%       disp(' Warning: x, y, and z have different sizes ?!')
%       result = zeros(size(x));
%end
```

```
function  [h1,h2] = func_h(x,y,z)
% usage:  [h1,h2] = func_h(x,y,z)
% where   x,y,z and h1,h2 are of the same size.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The observation function
%%      Chuanxia Rao
%%      April 1997
%%      May 1997
%%      Jul 1997
%%-----------------------------------------
%%      Called in:
%%      obs_cor.m
%%      generate.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if (size(x)==size(y) & size(y)==size(z)),
        [ll,mm,nn] = size(y);
for i=1:ll,
for j=1:mm,
for k=1:nn,
        temp12 = x(i,j,k)^2 + y(i,j,k)^2;
if temp12==0,
        h1(i,j,k) = 0;
        h2(i,j,k) = asin( sign(z(i,j,k)) );
else
        arg1 = x(i,j,k) / sqrt(temp12);
        temp1 = sign(y(i,j,k)) * acos(arg1);
        h1(i,j,k) = temp1;
        temp = temp12 + z(i,j,k)^2;
        arg2 = z(i,j,k) / sqrt(temp);
        h2(i,j,k) = asin(arg2);
end
end
end
end
else
        disp(' Warning: x, y, and z have different sizes ?!')
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Initial data for the cont.-discr. model:
%%      dX = b(X)dt + sigmaa*dW
%%      X(0) ~ N(m0,var0)
%%      z(k) = h(X(t_k)) + delta*v(k)
%%-----------------------------
%%      Chuanxia Rao
%%      Februay 1996
%%      Februry 1997
%%      July 1997
%%-----------------------------
%%      Called in:
%%      all other parts
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global  dim dim_obs
global  nx ny nz
global  xa ya za
global  xb yb zb
global  xx yy zz
global  xend0 yend0 zend0
global  xend1 yend1 zend1
global  m0 var0_inv

global  dt dt2
global  denom
global  denom_obs
%global nxp1 nyp1 nzp1

global substeps
global small %large
global resol_vert


        disp(' Initialization ...');
        dim = 3;
        dim_obs = 2;

Tfinal = 2.0;
Kfinal = 256; Kt = Kfinal;
Kmax = 150 %128; %64; %256;
%Kmax = Kfinal - 1;
%Kmax = Kfinal / 4;

nx = 12;
ny = 10;
nz = 8;
xa = 0.15;  xb = 0.75;
ya = 0.30;  yb = 0.60;
za = 0.14;  zb = 0.30;
%nx = 50;
%ny = 50;
%nz = 30;
%xa = -0.25;  xb = 0.75;
%ya = -0.40;  yb = 0.60;
%za = 0.0;  zb = 0.30;

xend0 = -0.95;  xend1 = 0.75;
yend0 = -0.60;  yend1 = 0.60;
zend0 = -0.30;  zend1 = 0.30;
%nx = 72; %60; %40; %80;
%ny = 60; %45; %30; %60;
%nz = 30; %24; %20; %30;
%xa = [-0.95, -0.6, -0.3];
```

```
%xb = [0.75, 0.6, 0.3];
%Nx = [nx,ny,nz];
%Lx = [3,2,1];
%Lx = [1,1,1];                  %% bandwidth of local propogation:
                                %% It depends on (dx*dx)/(dt*sigmaa^2).

plot_step = 2; %1;              %% plot in every plot_step steps
resol_vert = 0.01;              %% for plotting target position

substeps = 1;
small = 1e-16;                  %% threshold for positive probability
%large = -log(small);


%sigmaa = [0.0, 0.0, 0.0];      %% coefficients in signal noise
sigmaa = [0.045, 0.023, 0.012];
%delta = [0.0, 0.0];            %% coefficients in observ. noise
%delta = [0.24, 0.12];
delta = [0.64, 0.36];

m0 = [0.45, 0.42, 0.20];        %% initial mean
var0_inv = [100, 121, 64];      %% initial variance, its inverse
%var0_inv = [82, 100, 64];

%X0 = [0.75, -0.5, -0.25];
X0 = [3.6/4/2, 1./2, 1./4];
Z0 = [0,0];


%%
%%  Further initialization:
%%  -- no changes needed --
%%

dt = Tfinal/Kfinal;
dx = (xb-xa)/nx;
dy = (yb-ya)/ny;
dz = (zb-za)/nz;

dt2 = dt/2;
%dt4 = dt/4;
nxp1 = nx + 1;
nyp1 = ny + 1;
nzp1 = nz + 1;

denom_obs = 2 * delta.^2;               %% a row vector, not a diagonal matrix
denom = (2*dt) * sigmaa.^2;             %% a row vector
%const_norm = sigmaa... * sqrt(2*pi*dt)^dim;

disp(' Setting-up of initial density and observation function ...')
        xx = xa + dx * (0:nx);
        yy = ya + dy * (0:ny);
        zz = za + dz * (0:nz);
        pk = func_p0(xx, yy, zz);
%       pk = cutsmall(pk);

        x1d = xx;
        y1d = yy;
        z1d = zz;
        xx = zeros(nxp1,nyp1,nzp1);
        yy = xx;
        zz = yy;
for i=1:nxp1,
for j=1:nyp1,
```

```
for k=1:nzp1,
        xx(i,j,k) = x1d(i);
        yy(i,j,k) = y1d(j);
        zz(i,j,k) = z1d(k);
end
end
end

%       [hh1, hh2] = func_h(xx, yy, zz);
%%      hh1 = cutsmall(hh1);
%%      hh2 = cutsmall(hh2);
```

```
function  p0 = func_p0(x,y,z)
% usage:  p0 = func_p0(x,y,z)
% where   x,y,z are vectors.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The initial density function
%%      Chuanxia Rao
%%      Februry 1997
%%      May 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Called in:
%%      moreinit.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global m0 var0_inv

%       temp = sqrt(2*pi)^3;
%       temp0 = temp*sqrt(var0_inv(1)*var0_inv(2)*var0_inv(3));
for i=1:length(x),
        temp1 = (x(i)-m0(1))^2 * var0_inv(1);
for j=1:length(y),
        temp2 = (y(j)-m0(2))^2 * var0_inv(2);
for k=1:length(z),
        temp3 = (z(k)-m0(3))^2 * var0_inv(3);
        temp3 = temp1 + temp2 + temp3;
        p0(i,j,k) = exp( -temp3/2 );
end
end
end
%       p0 = p0 / temp0;
        p0 = p0 / max( max( max(p0,[],3),[],2 ),[],1 );
```

```
function  [b1,b2,b3] = func_b(x,y,z)
% usage:  [b1,b2,b3] = func_b(x,y,z)
% where   x,y,z and b1,b2,b3 are of the same size.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The signal propogation function
%%      Chuanxia Rao
%%      Februry 1997
%%      May 1997
%%----------------------------------------
%%      Called in:
%%      generate.m
%%      advection.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%for j=1:length(y),
%for k=1:length(z),
%%      b1(1:length(x), j, k) = 200*y(j)^3 + 50*z(k) - 50;
%       b1(1:length(x), j, k) = - 200*y(j)^3 + 50*z(k);
%end
%end
%       b1 = 200*y.^3 + 50*z - 50;
        b1 = -200*y.^3 + 50*z;
        b1 = b1/2;

%       b2 = 1./2 * ones(size(b1));
        b2 = -1./2 * ones(size(b1));

%       b3 = 1./4 * ones(size(b1));
        b3 = -1./4 * ones(size(b1));

%       b = [b1, b2, b3];
```

```
function  result = func_db(x,y,z)
% usage:  result = func_db(x,y,z)
% where   x,y,z, and result are of the same size.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The scalar function  Delta * b
%%      Chuanxia Rao
%%      July 1997
%%      May 1997
%%----------------------------------------
%%      Called in:
%%      prior.m (& onestep.m)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if (size(x)==size(y) & size(y)==size(z)),
        result = zeros(size(z));
else
        disp(' Warning: x, y, and z have different sizes ?!')
        result = zeros(size(x));
end

%n1 = length(x);
%n2 = length(y);
%n3 = length(z);
%       db1 = zeros(n1,n2,n3);
%       db2 = zeros(n1,n2,n3);
%       db3 = zeros(n1,n2,n3);
%n = length(x) * length(y) * length(z);
%       db1 = zeros(n,1);
%       db2 = zeros(n,1);
%       db3 = zeros(n,1);
%       db = zeros(n,1);

%%      db = db1 + db2 + db3;
%%      db1 = db / 3;
%%      db2 = db1;
%%      db3 = db1;
```

```
function  [h1,h2] = func_h(x,y,z)
% usage:  [h1,h2] = func_h(x,y,z)
% where   x,y,z and h1,h2 are of the same size.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The observation function
%%      Chuanxia Rao
%%      April 1997
%%      May 1997
%%      Jul 1997
%%----------------------------------------
%%      Called in:
%%      obs_cor.m
%%      generate.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if (size(x)==size(y) & size(y)==size(z)),
        [ll,mm,nn] = size(y);
for i=1:ll,
for j=1:mm,
for k=1:nn,
        temp12 = x(i,j,k)^2 + y(i,j,k)^2;
if temp12==0,
        h1(i,j,k) = 0;
        h2(i,j,k) = asin( sign(z(i,j,k)) );
else
        arg1 = x(i,j,k) / sqrt(temp12);
        temp1 = sign(y(i,j,k)) * acos(arg1);
        h1(i,j,k) = temp1;
        temp = temp12 + z(i,j,k)^2;
        arg2 = z(i,j,k) / sqrt(temp);
        h2(i,j,k) = asin(arg2);
end
end
end
end
else
        disp(' Warning: x, y, and z have different sizes ?!')
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Generate a 3D trajectory for
%%      dX = b(X)dt + sigmaa*dW
%%      X(0) = X0
%%      And generate an observation for
%%      z(k) = h(X(t_k)) + delta*v(k)
%%      Chuanxia Rao
%%      January 1996
%%      January 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Dependencies:
%%      initialize.m;
%%      func_b.m (function)
%%      func_h.m (function)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Output:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       initialize;
        disp(' Generating the trajectory ...');

%out_fileZ = 'data_Z.m';
%out_fileX = 'data_X.m';

%Kfinal = Kt;
srdt = sqrt(dt);
X(1,:) = X0;
Z(1,:) = Z0;

        randn('seed', 0);
        v = randn(Kfinal,2);

        Xk = X(1,:);
for k = 1:Kfinal,
        k1 = k + 1;

%       [bk1,bk2,bk3] = func_b(Xk(1),Xk(2),Xk(3));
%       bk = [bk1,bk2,bk3];
%       temp = Xk + dt*bk;              %% forward Euler
%       [bk1,bk2,bk3] = func_b(temp(1), temp(2), temp(3));
%       b2 = bk + [bk1,bk2,bk3];
%       temp = Xk + dt2 * b2;           %% trapezoid
%       temp1 = srdt*diag(sigmaa)*randn(dim,1);
%       X(k1,:) = temp + temp1';
        [x1,x2,x3] = advection(dt, Xk(1),Xk(2),Xk(3));
        temp1 = srdt * sigmaa .* randn(1,dim); %randn(size(sigmaa));
        X(k1,:) = [x1,x2,x3] + temp1;
%       [T,Y] = ode23('func_fb', [0 dt], [Xk(1),Xk(2),Xk(3)]);
%       [mY,nY] = size(Y);
%       X(k1,:) = Y(mY,:) + temp1;

        Xk = X(k1,:);
        [h1,h2] = func_h(Xk(1),Xk(2),Xk(3));
        temp2 = diag(delta)*v(k,:)';
        Z(k1,:) = [h1,h2] + temp2';
end

%%
%%      Output -- graph:
%%

        k1 = 1:(Kfinal+1);
        k11 = (1:Kfinal)+1;
        t1 = 0:dt:Tfinal;
```

```
t11 = dt:dt:Tfinal;
a1 = min(X(k1,1)); b1 = max(X(k1,1));
a2 = min(X(k1,2)); b2 = max(X(k1,2));
a3 = min(X(k1,3)); b3 = max(X(k1,3));
aZ1 = min(Z(k11,1)); bZ1 = max(Z(k11,1));
aZ2 = min(Z(k11,2)); bZ2 = max(Z(k11,2));

figure;
subplot(221), plot(t1,X(k1,1));
axis([0,Tfinal, a1,b1]);
subplot(222), plot(t1,X(k1,2));
axis([0,Tfinal, a2,b2]);
subplot(223), plot(t1,X(k1,3));
axis([0,Tfinal, a3,b3]);
subplot(224), plot3(X(k1,3), X(k1,1), X(k1,2), 'r');
axis([a3,b3, a1,b1, a2,b2]);

figure;
subplot(211), plot(t11,Z(k11,1));
axis([0,Tfinal, aZ1,bZ1]);
subplot(212), plot(t11,Z(k11,2));
axis([0,Tfinal, aZ2,bZ2]);

clear temp temp1 temp2
clear bk bk1 bk2 bk3
clear h1 h2
clear v srdt
clear a1 a2 a3 aZ1 aZ2
clear b1 b2 b3 bZ1 bZ2
clear k1 k11
clear t1 t11
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      computation of the UFD for the filtering problem:
%%      dX = b(X)dt + sigmaa(X)dW,  X(0) ~ N(m0,var0)
%%      Z(k) = h(X(t_k)) + delta(k)v(k)
%%-------------------------------------------------
%%      Chuanxia Rao
%%      Jul 1997
%%      Mar 1997
%%      Feb 1996
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Dependencies:
%%      initialize.m
%%      moreinit.m
%%      obs_cor.m
%%      prior.m
%%      cutsmall.m
%%      generate.m (for observations Z)
%%-------------------------------------------------
%%      Output:
%%      posterior density pk1 (denoted as pk)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%initialize;
%%moreinit;
        %% -- They must be called before running this file.

        [xx,yy,zz, pk] = prior(xx,yy,zz, pk);
%       Tpk = cutsmall(Tpk);
                                                %% -- prior density Tpk
        alpha_k1 = obs_cor(Z(k1,1),Z(k1,2), xx,yy,zz);
%       alpha_k1 = cutsmall(alpha_k1);
                                                %% -- corrector alpha_k1
%       pk = alpha_k1 .* Tpk;
        pk = alpha_k1 .* pk;
        pk_max = max( max( max(pk,[],3),[],2 ),[],1 );
        if pk_max > 0,
        pk = pk/pk_max;
        end
        pk = cutsmall(pk);
                                                %% -- density at step k1
%clear  alpha_k1 Tpk
clear  alpha_k1
```

```
function  [x_new,y_new,z_new, p_new] = prior(x_old,y_old,z_old, p_old)
% usage:  [x_new,y_new,z_new, p_new] = prior(x_old,y_old,z_old, p_old)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Chuanxia Rao
%%      July 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Dependencies:
%%      advection.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global  dt dt2 denom

        [x_new, y_new, z_new] = advection(dt, x_old, y_old, z_old);

        temp = - func_db(x_new, y_new, z_new);
        temp = temp - func_db(x_old, y_old, z_old);
        temp = dt2 * temp;
        p_new = exp( temp );
        p_new = p_old .* p_new;

        [m1,m2,m3] = size(p_new);
        m1m1 = m1-1;
        m2m1 = m2-1;
        m3m1 = m3-1;

        x_old = x_new(2:m1,:,:);
        temp = exp(-(x_old-x_new(1:m1m1,:,:)).^2/denom(1));
        temp = temp .* p_new(2:m1,:,:);
        p_new(1:m1m1,:,:) = p_new(1:m1m1,:,:) + temp;
        x_old = x_new(1:m1m1,:,:);
        temp = exp(-(x_old-x_new(2:m1,:,:)).^2/denom(1));
        temp = temp .* p_new(1:m1m1,:,:);
        p_new(2:m1,:,:) = p_new(2:m1,:,:) + temp;

        y_old = y_new(:,2:m2,:);
        temp = exp(-(y_old-y_new(:,1:m2m1,:)).^2/denom(2));
        temp = temp .* p_new(:,2:m2,:);
        p_new(:,1:m2m1,:) = p_new(:,1:m2m1,:) + temp;
        y_old = y_new(:,1:m2m1,:);
        temp = exp(-(y_old-y_new(:,2:m2,:)).^2/denom(2));
        temp = temp .* p_new(:,1:m2m1,:);
        p_new(:,2:m2,:) = p_new(:,2:m2,:) + temp;

        z_old = z_new(:,:,2:m3);
        temp = exp(-(z_old-z_new(:,:,1:m3m1)).^2/denom(3));
        temp = temp .* p_new(:,:,2:m3);
        p_new(:,:,1:m3m1) = p_new(:,:,1:m3m1) + temp;
        z_old = z_new(:,:,1:m3m1);
        temp = exp(-(z_old-z_new(:,:,2:m3)).^2/denom(3));
        temp = temp .* p_new(:,:,1:m3m1);
        p_new(:,:,2:m3) = p_new(:,:,2:m3) + temp;

%       [pmax, i,j,k] = max(p_new);
%       p_new = p_new / pmax;
%       star = [x_new(i,j,k), y_new(i,j,k), z_new(i,j,k)];
%       x_star = [x_star' star']';

%return
```

```
function  result = obs_cor(z1,z2, x,y,z)
% usage:  result = obs_cor(z1,z2, x,y,z)
% where   z1,z2 are the measurements, and
%         (x,y,z) are the spatial points.
%         %hh1, hh2 are the noise-free part.
%         dim = 3; dim_obs = 2;

%%      Chuanxia Rao
%%      July 1997
%%      May 1997
%%      Feb 1996
%%      Used in: onestep.m

global denom_obs        %% defined in initialize.m

        [hh1, hh2] = func_h(x, y, z);
        temp = z1 - hh1;
        temp1 = temp/denom_obs(1) .* temp;
        temp = z2 - hh2;
        temp1 = temp1 + temp/denom_obs(2) .* temp;
        temp1 = temp1 - min( min( min(temp1,[],3),[],2 ),[],1 );
        result = exp( - temp1 );
```

```
function  xyz = peak(pk)
%usage:   xyz = peak(pk)
%         where  size(pk) = [nxp1 nyp1 nzp1]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%       This is for plotting the tracking process.
%%       Chuanxia Rao
%%       April 1997
%%       May 1997
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global  xx yy zz

        [one, ijk] = max(pk,[],3);
        [one, ij] = max(one,[],2);
        [one, i] = max(one,[],1);
        j = ij(i);
        k = ijk(i,j);
        x = xx(i,j,k);
        y = yy(i,j,k);
        z = zz(i,j,k);
        xyz = [x,y,z]';

return
```

C.Rao
advection.m


```
function  [x_new,y_new,z_new] = advection(dt0, x_old,y_old,z_old)
% usage:  [x_new,y_new,z_new] = advection(dt0, x_old,y_old,z_old)

%%      Chuanxia Rao
%%      Februry 1997
%%      April 1997
%%      June 1997

global  substeps

dt = dt0/substeps;
%dt6 = dt/6;
dt2 = dt/2;
x = x_old;
y = y_old;
z = z_old;

for k = 1:substeps,
        [bk1,bk2,bk3] = func_b(x,y,z);
        [b1,b2,b3] = func_b(x+dt*bk1,y+dt*bk2,z+dt*bk3);
        x = x + dt2 * (bk1 + b1);
        y = y + dt2 * (bk2 + b2);
        z = z + dt2 * (bk3 + b3);       %% trapezoid
%       bk = func_b(x);
%       temp = func_b(x + dt2*bk);
%       x = x + dt * temp;              %% modified Euler
%%      k1 = func_b(x);
%%      k2 = func_b(x + dt2*k1);
%%      k3 = func_b(x + dt2*k2);
%%      k4 = func_b(x + dt*k3);
%%      temp = k1 + 2*k2 + 2*k3 + k4;
%%      x = x + dt6 * temp;             %% Runge-Kutta-4
end
        x_new = x;
        y_new = y;
        z_new = z;
```

```
function  result = cutsmall(array)
% usage:  result = cutsmall(array)

%%      Chuanxia Rao
%%      May 1997
%%      Used in: onestep.m

global small            %% defined in initialize.m

        result = (array > small) .* array;
%       result = sparse( result );
```

```
function  plotting(xyz, pk)
%usage:   plotting(xyz, pk)
%         where length(xyz) = dim(=3)
%               size(pk) = [nxp1 nyp1 nzp1]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%       This is for plotting the tracking process.
%%       Chuanxia Rao
%%       April 1997
%%       May 1997
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global  xend0 yend0 zend0
global  xend1 yend1 zend1
global  xx yy zz
global  resol_vert

        x = xyz(1);
        y = xyz(2);
        z = xyz(3);

        plot3(z, x, y, 'r*');
        axis([zend0 zend1 xend0 xend1 yend0 yend1]);
%       axis([xend0 xend1 yend0 yend1 zend0 zend1]);
%       hold on;

        [one, ijk] = max(pk,[],3);
        [one, ij] = max(one,[],2);
        [one, i] = max(one,[],1);
        j = ij(i);
        k = ijk(i,j);
        x = xx(i,j,k);
        y = yy(i,j,k);
        z = zz(i,j,k);
%       plot3(x, y, z, 'ro', 'LineWidth',[2]);
        plot3(z, x, y, 'g*');
        grid on;
        drawnow; %pause(1);
%       hold off;

return
```